Some Design Issues for High-speed Networks

Van Jacobson

van@ee.lbl.gov
Lawrence Berkeley Laboratory
Berkeley, CA 94720

Networkshop '93
Melbourne, Australia
30 November, 1993

Outline:

- Can the Internet protocols deal with high speed?
- Do we need any additions to the architecture?

Can the Internet protocols deal with high speed?

Yes.

How do we know this?

We've done a ground-up re-implementation of the BSD Unix kernel network architecture based on new understanding gained during five years of measuring and tuning the old code.

The new code is slightly faster than the old (one to two orders of magnitude).

Was going to be released as part of 4.4BSD. Demise of CSRG and AT&T suit made that impossible. Trying to work out how to finish code and do distribution.

How fast do things go?

Measurements (made with external logic analyzer) of current prototype running on a Sparcstation-2 show:

- The total cost for IP to forward a packet is 37 instructions and 7 memory references
  (5 loads and 2 stores).

- The total cost to process the protocol in a TCP datagram is <60 instructions and 22 memory references.

- For any reasonable datagram size, TCP/IP runs *application-to-application* at the speed of main memory.

- Poor performance is almost always the fault of baroque, complex, poorly integrated (i.e., typical) network interface hardware.

What's Different?

Prototype is almost the same as BSD Net-2 with a few minor changes:

- mbufs are gone.
- sockbufs are gone.
- netipl is gone.
- sosend / soreceive are gone.
- inpcb's are gone.
- pr_usrreq is gone.
- ip_output, and most of the protocol layering, are gone.
- . . .

What was wrong with old code?

Short answer: Layering.

- Old code's strict layering model caused bad buffering decisions which generated lots of extra memory traffic. New code touches any piece of data exactly once.

- Old code's layering model lost perfomance due to lack of parallelism. New code dumps layering and maximizes parallelism.

Layering is bad?

Dave Clark at MIT (and many others) have been saying for the past decade that layered models are a great way to design protocols but a lousy way to implement them. All our measurements support this theory. So ...

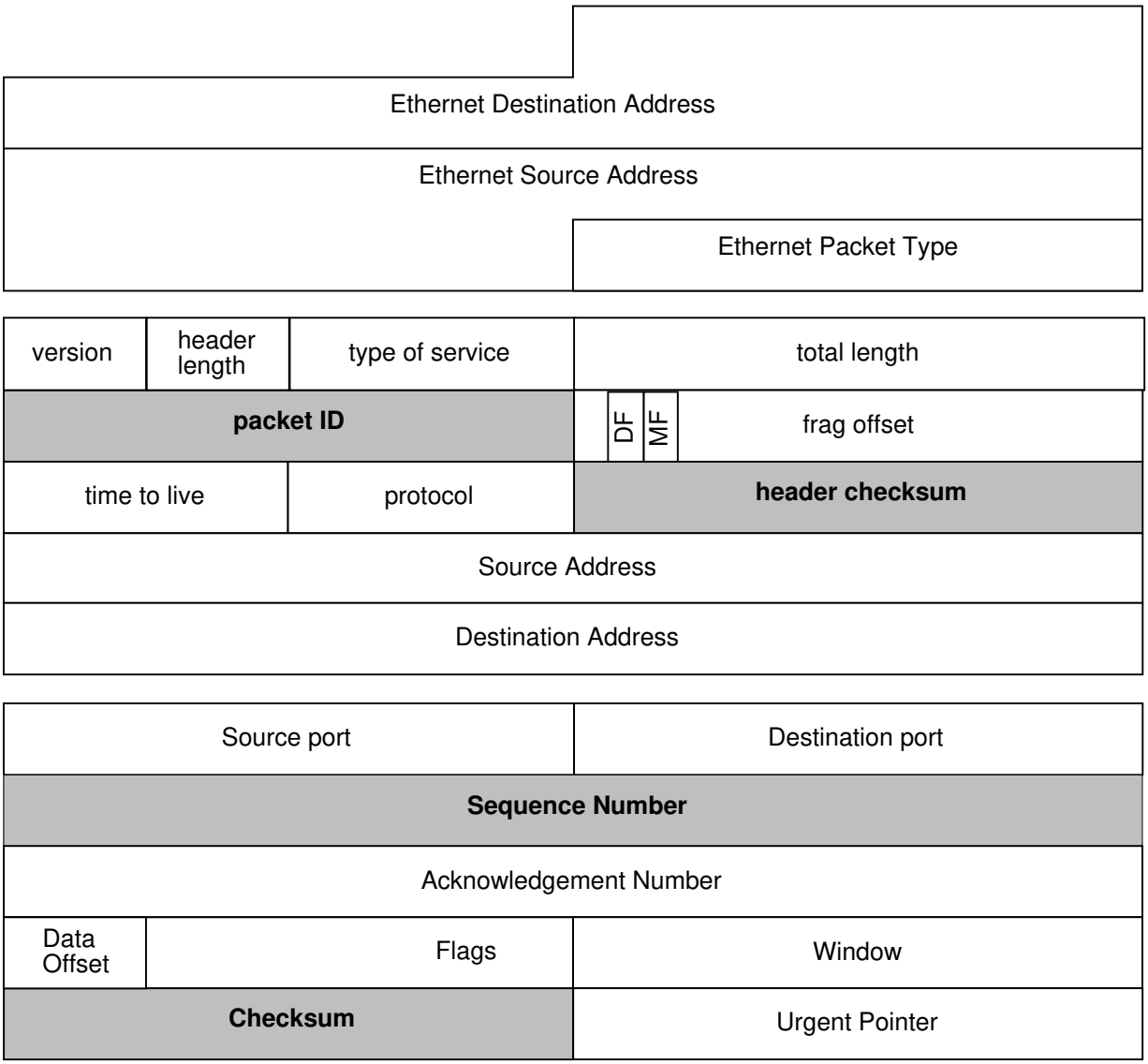
Application level send and receive go to protocol specific routines, not generic routines, that know what packets look like and understand protocol's flow control and reliability policy.

These routines call output driver, not generic allocator, to get packet buffers.

They build and ship one packet at a time to maximize parallelism.

They cache everything that might be useful later and can do most of their packet building with a single bcopy.

TCP/IP Packet Changes
(data packets)



But what about the checksum?

If you architect things right, it costs nothing.

Copy and Checksum Timings
(all times in ns/byte)

Machine	*bcopy*	*bcopy & cksum*	*cksum cost*
Sun 3/60	133	206	58%
HP9000/370	(68) 122	(97) 144	(43) 18%
Sparcstation-1	(94) 164	(108) 177	(15) 8%
Sparcstation-2	(42) 109	(49) 109	(15) 0%
HP9000/720	(20) 54	(20) 54	(0) 0%

(numbers in paren are for all data in cache)

Where does the time go?

Say we're receiving back-to-back 4KB FDDI packets (one packet every 330 μs).

On a Sparcstation-2 (40MHz clock or 25ns/instr.):

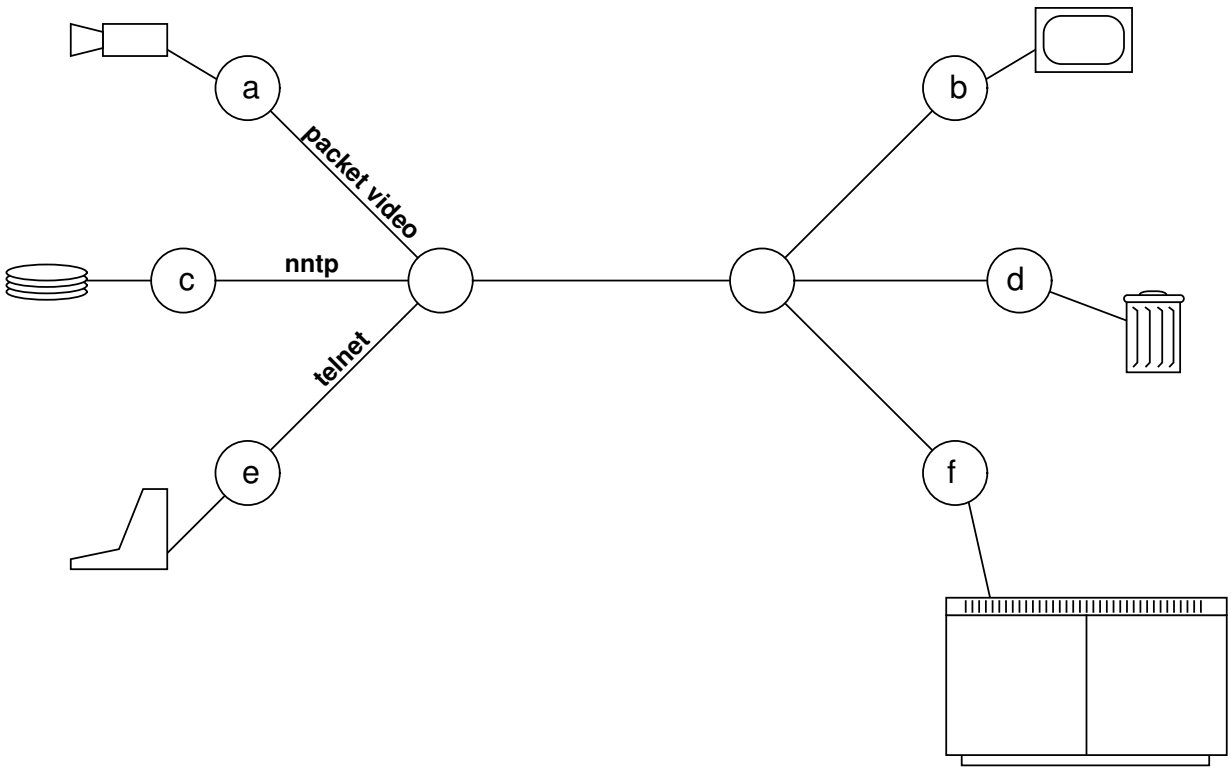
	instr	μ *s*
TCP + IP + ARP	100	3
Interrupt entry/exit	600	25
DMA 4KB into memory	–	164
copy 4KB to application	1000	446

Architectural Addition:

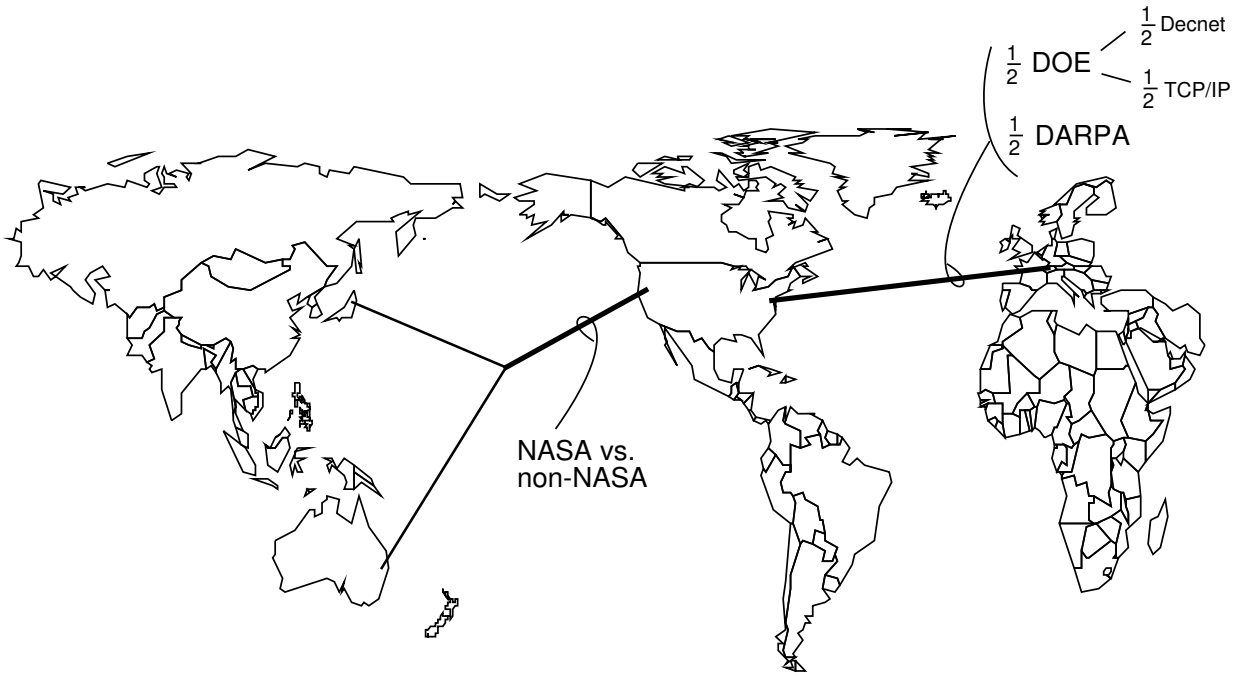
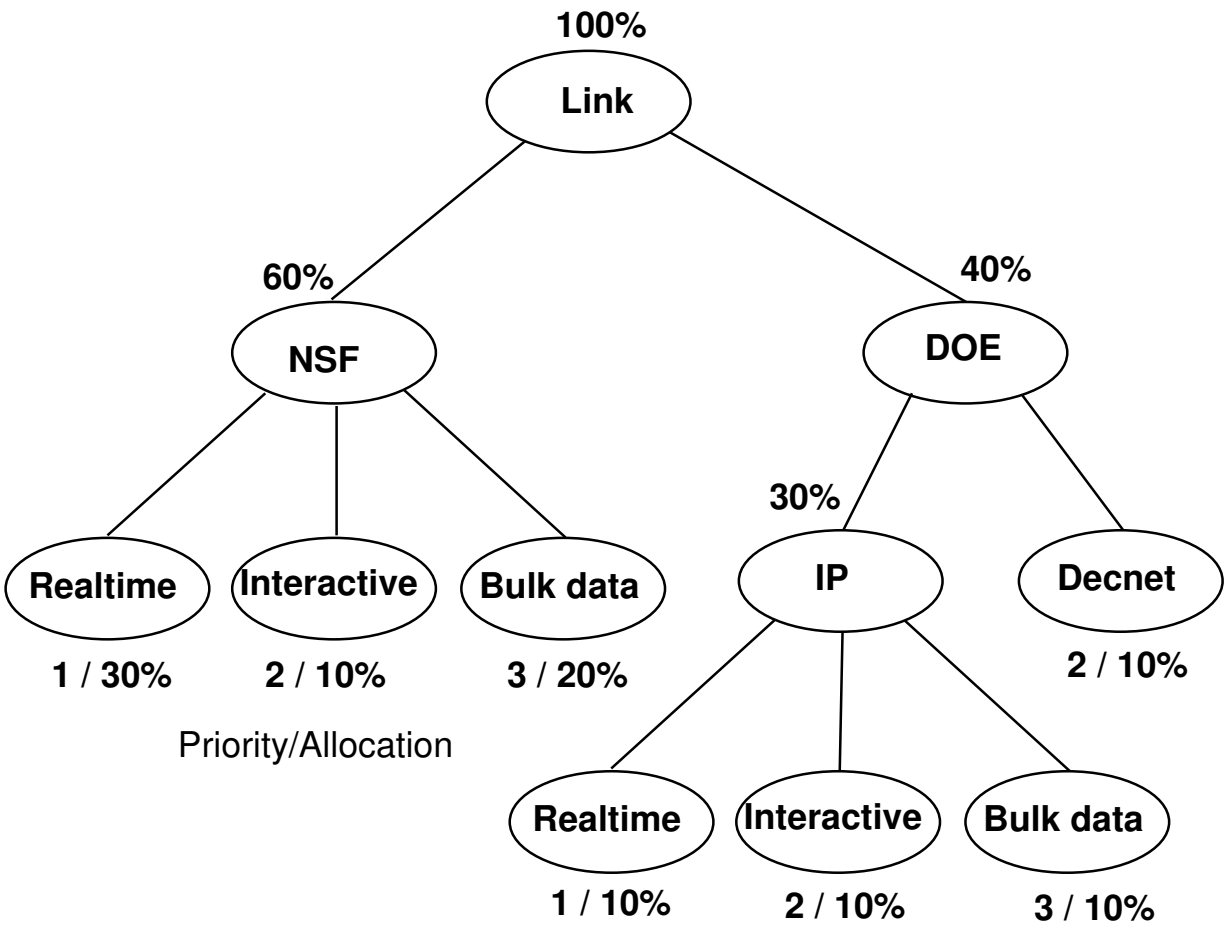
As links get fatter we (can) get much wider range of applications and larger mix of users simultaneously filling pipe.

We need better control of how bandwidth is portioned out.

Need to support different qualities of service (real-time vs. interactive vs. bulk data)



Need to accommodate ‘policy’ constraints. E.g., DOE & DARPA fund half of US–UK link. Each wants half if link loaded, wants excess if other guy isn’t using all his half. DOE also needs to divide link between DECNet & IP protocols using similar rules.



Combining policy with QOS suggests that traffic must satisfy a hierarchy of constraints: